YADA - Yet Another Debianisation Aid

YADA is a Debian packaging helper. It parses a special file, `debian/packages`, in a source package, and generates an appropriate `debian/rules`to control building of the package.

1. How YADA works

The basic idea is very simple: everything that used to be scattered amongst many little files in your `debian/` directory is now concentrated into a single file, `debian/packages`. There are only a couple of exceptions: `debian/changelog`is unchanged, and `debian/yada`is the YADA script, which you must copy into `/usr/bin` or into your `debian/` directory. You can do this with the command **yada** `yada`. `debian/rules`, `debian/control`and optional `debian/templates` are now generated from `debian/packages`by YADA. Most of the other files there will then likely be redundant.

So the only thing you now need to know to switch to YADA is how to write `debian/packages`! Read on.

When you've written `debian/packages`, you'll need to run **yada** `generate` in order to generate your new `debian/control`and `debian/rules`. After that, your rules file should automatically regenerate both itself and `debian/control`as necessary.

2. yada command

YADA is a just perl script so it can be installed globally in the system in `/usr/bin` directory or can be placed in local `debian/` directory. It is also possible to use newer yada script in `debian/` directory than installed in `/usr/bin` directory.

Commands available:

- **yada** `yada`

  This command creates an skeleton `debian/packages`file for you to fill in, if you don't already have one and creates `debian/changelog`for initial release.

- **yada** `rebuild rules`

  YADA reads `debian/packages`and generates a new rules file, `debian/rules`. Note that any existing rules file will be overwritten, and no backup will be kept.

- **yada** `rebuild control`

  YADA reads `debian/packages`and generates a new control file, `debian/control`. Note that any existing control file will be overwritten, and no backup will be kept.

- **yada** `rebuild templates`

  YADA reads `debian/packages`and generates a new optional templates file, `debian/templates`. Note that any existing templates file will be overwritten, and no backup will be kept. This file is created only if `debian/packages`contains any Templates: field. The file might be used with debconf-updatepo(1) command of po-debconf(7) system to regenerate DebConf translations located in `debian/po/` directory.

- **yada** `rebuild`

  Regenerates all above required files if they don't exists already.

3. The format of `debian/packages`

`debian/packages`is based on the format of `debian/control`, but with several differences. I'll explain the format from scratch.

`debian/packages`is formed from a series of paragraphs, separated by blank lines. Empty paragraphs are ignored, so you can add extra blank lines before and after paragraphs without problems. ALL lines are stripped of trailing whitespace, in order to ensure that what you see is what YADA sees. (I'm paranoid about trailing whitespace.)

Lines beginning with a hash mark ("#") at the left margin are ignored completely. If the hash mark has white space in front of it, the line is treated as part of an extended field if appropriate; if not, it is ignored.

Lines beginning with a percent mark ("%") at the left margin means macro commands similar to macros used by C preprocessor. The defined macro variables can be further used in common paragraphs or as parameters for other macro commands.

Each paragraph is made up of fields, each of which associates a keyword with a textual value. A field's value can be single-line or multi-line. The first (or only) line of a field starts at the left margin with a case-insensitive keyword containing alphanumerics and hyphens, followed by a colon, followed by the first (or only) line of the field's value. Subsequent lines of the field start with a space character at the left margin, and are followed by one line of the field's value.

Here are a couple of example paragraphs in this format:

```
Word: gnu
Part-Of-Speech: noun
# Note to myself: must fix this pronunciation
Pronunciation: guh-NOO
Definition: a large animal in the
 antelope family, which has a hairy
 coat.

Word: gnat
Part-of-speech: noun
Definition: a small insect which bites
 anything that moves.
```

The observant will have noticed that this leaves no way to include a blank line in a field's value; since trailing whitespace is stripped, a line containing only a space would be treated as the end of the paragraph. There is an escape sequence for this: a line containing a single dot (a.k.a. full stop or period) after the initial space will be treated as blank.

In fact, any line containing only dots after that initial space will have one of them stripped off before being processed. Lines starting with a space and a dot, but which contain a character other than a dot anywhere in the line are left unmolested.

So, we can include blank lines like this:

```
Dish:           Boiled lobster
Ingredients:
 1 lobster
 1 anvil
 1 saucepan
Method:
 First, catch your lobster.
 .
 When you have it cornered, stun it
 by hitting it over the head with the
 anvil, then quickly put it into the
 saucepan and boil it.
 .
 You should take great care not to let
 the lobster take posession of the
```

```
 anvil; a lobster with an anvil can
 make your life hell.
```

That example also demonstrates another minor feature: blank space is stripped from the beginning of the first line of a multi-line value. If that first line is entirely white space, the whole line is ignored, and the value starts on the line AFTER the line containing the keyword.

4. How to write `debian/packages`

There are two kinds of paragraph in `debian/packages`. The first paragraph in the file describes the source package, describing how to build it, how to clean it, what it's called, where it came from, who maintains it, etc. The following paragraphs each describe a binary package which can be built from the source package.

4.1. Merged fields

YADA merges several fields with the same keyword into one field. So, if we have several "Postinst: ..." sections, they will be concatenate (or a error will be generated if different shells are requested). There are also added two general keyword: "After-" and "Before-" that can be prepend to the classical keywords. With this, these section are concatenate after or before the regular section.

In example, the section:

```
Build: sh
 echo test build
After-Build: sh
 echo test after-build
Before-Build: sh
 echo test before-build
```

will be concatenated into one field:

```
Build: sh
 echo test before-build
 echo test build
 echo test after-build
```

4.2. Executable fields

Several fields contain commands to be executed at appropriate points during the processing of the package. The first line of one of these executable fields specifies which command processor is to be used to execute the field; subsequent lines are the commands to be executed.

At present, the only command processor recognised by YADA is **sh** and **bash**, the bourne shell. The rest of the field is interpreted as a shell script fragment. The fragment will be executed with the shell's `-e` option set, so that if any command fails, the whole script will fail.

```
Source: libxyz
 [...]
Build: bash
 ./configure --prefix=/usr
 CC=${CC:-gcc}
 CFLAGS=${CFLAGS:--Wall -g}
 if [ "${DEB_BUILD_OPTIONS#*noopt}" != "$DEB_BUILD_OPTIONS" ]; then
     CFLAGS="$CFLAGS -O0"
 else
     CFLAGS="$CFLAGS -O2"
 fi
 make CFLAGS="$CFLAGS" LDFLAGS="$LDFLAGS" CC="$CC"
Clean: sh
 make distclean || true
```

In some cases, extra variables or commands may be available for use by an executable field. These are described below.

4.3. Environment variables

Several extra environment variables are available to use with Build, Install, Finalize, Preinst, Postinst, Prerm, Postrm, Config and Init fields.

`ROOT`

The root of the temporary filesystem image to install into. You won't need to use this in most cases.

`TMPROOT`

The temporary directory shared with all binary packages. It can be used for **make** `install DESTDIR=$TMPROOT`.

`CONTROL`

The directory into which control files are to be installed. You probably won't need to use this unless you install control files that yada doesn't already know about.

`PACKAGE`

The name of the binary package being built. This variable is set only for binary packages.

`SOURCE`

The name of the source package being built.

`VERSION`

The version of the Debian source package.

`DEB_*`

The variables will be set by dpkg-architecture(1) command. You will need these variables if you use different settings for various architectures (i.e. special optimalization for i386 or alpha).

4.4. Fields in the source paragraph

The following fields have the same meaning as they do in `debian/control`, and should all be present in `debian/packages`:

Source

The name of the source package.

Section

The section (main, contrib, non-free or non-us) and subsection (admin, devel, games, x11, etc.) of the source package in the archive, separated by a forward slash. If the section is main, give only the subsection.

Priority

How necessary the programs or data contained in source package are to the running of the system (required, important, standard, optional or extra).

Maintainer

The full name and email address of the person currently responsible for this source package. The email address should be separated from the name by a single space, and surrounded by angle-brackets.

Standards-version

Which version of Debian policy the maintainer believes this package conforms to.

Origin

This field changes origin of the package. The default value is "debian" which means the information about origin is in the file `/etc/dpkg/origins/debian`.

Bugs

The URI for Bugs Tracking System, i.e. "debbugs://bugs.debian.org" for packages from Debian distribution.

Build-Depends

Build-Depends-Indep

These fields specify relationships between this package and other packages at build time. They each comprise a comma-separated list of dependencies, which are treated as set out in the Debian packaging manual. A dependency is either a single package, a list of alternative packages separated by vertical bars. The YADA automatically adds build depends on "file, patch, perl" if called `debian/yada` or "yada" if called `/usr/bin/yada`.

The following fields are defined by YADA, and you should use them in `debian/packages`. YADA uses the first four of these to construct the `/usr/share/doc/`*`package`*`/copyright` file, so it is important they are correct.

Upstream-Source

The URI of the upstream source code, in the standard URI format surrounded by angle-brackets. (Note that a URL can be turned into a URI by prefixing it with "URL:".) If this field is not present, YADA will assume that the package is a Debian-native package.

Copyright

The first line of this field gives the names of the standard copyright licence which applies to this package, if any. The following lines should contain a copy of the source package's copyright notice and copyright licence. If any of the standard licences are mentioned, you need not write where their full text can be found on a Debian system; yada will add that information for you. The standard licence names defined at present are "GPL", "LGPL", "Artistic" and "BSD". If none apply, place a single dot on the first line, and include the complete copyright and licence notice.

Major-Changes

If any major changes have been made to the upstream source, list them here. This fulfils the Debian policy requirement that changes be listed, and fulfils the legal requirements of several common copyright licences.

Packaged-For

The name of the project or organisation for which you produced the package. For packages produced by registered Debian developers, this field should read "Debian". For others, it might read, for example, "GNU", or "Hungry Programmers", or "Corel Corp.". If you didn't create the package as part of your work for anyone other than yourself, then don't include a "Origin" field.

Description

The first line of this field gives the human-readable name of the package. For example, if the Source field reads "libc6", the first line of the Description field might read "The GNU C library, version 2".

The rest of the field should contain any descriptive text which pertains to ALL the binary packages this source package produces. It will be prepended to the Description field of each binary package, followed by a blank line.

Build

An executable field describing how to build the software contained in the package. One extra command is available in this field:

- yada fixup libtool [<pathname>]

  Performs the fixups described in Lintian's "libtool-workarounds.txt" to prevent libtool hardcoding shared library directories into binaries. This should be called AFTER the configure script has generated libtool, but before libtool gets used. If the libtool script is not named "libtool" in the current directory, specify its <pathname>.

This field is called with "build" target.

Build-Arch

An executable field describing how to build architecture depended code in the package. This field is called with "build-arch" and "build" targets.

Build-Indep

An executable field describing how to build architecture independed code in the package. This field is called with "build-indep" and "build" targets.

Clean

An executable field describing how to reverse the effects of the Build field. There are no extra commands or variables available. This field is called with "clean" target.

These fields are also defined by YADA, but you may not need to use them:

Home-Page

The URI of the World-Wide Web home page of the upstream package, in angle-brackets. You should include this if possible.

Upstream-Authors

The names and email addressed of upstream authors.

Packager

The name and email address of the person who originally created Debianised this package, if not the current maintainer.

Other-Maintainers

The names and email addresses of any previous maintainers of this package, excluding the original packager and the current maintainer.

Patches

A wildcard matching those files in the debian/ directory which should be treated as patches, and automatically patched into the source. It means that "yada patch ..." command is called before Build script and "yada unpatch" command is called after Clean script. Automatic patching is not activated unless you specify this field. This feature was inspired by the "*.dpatch" system in the egcs packages.

Most often, this field would be used like this:

```
Patches: *.diff
```

The matchings can be separated with whitespace:

```
Patches: patches/*-all.diff patches/*-i386.diff
```

The second example is equivalent of:

```
Build: sh
 yada patch "debian/patches/*-all.diff"
 yada patch "debian/patches/*-i386.diff"
```

Basically, it works as follows. Instead of applying patches to the source tree directly, and letting dpkg-source handle them, you place the patches in files in your `debian/` directory. The names of these files should be matched by the contents of the "Patches:" field; this is how yada recognises patch files. So, for example, if you are sent an optimiser patch for your compiler, you can simply copy the email to `debian/optimiser.diff`.

When your source package is built, each patch is applied to the source tree. When your package is cleaned, the patches are unapplied. Yada takes some care to keep track of the status (applied or not) of every patch using files named like `debian/patch-*-applied`, and it applies and unapplies the patches as necessary. (For safety's sake, you should make sure your pattern cannot match files of the form `patch-*-applied`.)

Often, patches are intended to patch files in subdirectories. This means that **patch** needs to be given the `-p<n>` option to tell it how many pathname components to strip from filenames. You can give options to **patch** by putting a `PATCHOPTIONS` line in the patch file. The line must contain the text `#PATCHOPTIONS:` at the start of a line. The rest of the line gives options which will be passed to **patch** when applying or unapplying that patch file.

4.5. Fields in binary paragraphs

The following fields can be used in the paragraphs describing binary packages. First, the fields which have the same meaning as in `debian/control`:

Package

The name of the binary package.

Version

If this field in binary package's paragraph exists, the package can contain different version number than source package version.

Architecture

The architecture(s) for which this binary package may be built. "all" means that it is architecture-independent; "any" means that it is not architecture-independent, but may be built for any architecture. "none" is a YADA extension and means that this binary package will never be built (useful for "commenting out" binary packages). Macros "linux", "hurd", "darwin", "freebsd", "netbsd" and "openbsd" are expanded to native dpkg's architecture names.

```
Package: package-linuxonly
Architecture: linux

Package: package-doc
Architecture: all

Package: package-intelonly
Architecture: i386
```

Section

Priority

Analogous to the Section and Priority fields in the source paragraph, these classify the binary package.

Essential

If "yes", **dpkg** will prevent the end-user from removing the binary package from the system unless `--force-remove-essential` is specified. Do not use this unless you have discussed it on debian-devel and the concensus opinion is that you may.

Pre-Depends, Depends, Recommends, Suggests, Enhances

These fields specify relationships between this package as other packages which may be installed on the target system. They each comprise a comma-separated list of dependencies, which are treated as set out in the Debian packaging manual. A dependency is either a single package, a list of alternative packages separated by vertical bars, or (a YADA extension) one or more filenames or file-globs in square brackets. Filenames in square brackets should be absolute filenames on the

installed system, and are fed to **dpkg-shlibdeps** to be analysed for shared library dependencies. The output of **dpkg-shlibdeps** is substituted for the square brackets and the file-globs before the field is placed into the binary package.

For example, most packages containing ELF binaries will use the line:

```
Package: mypackage
Depends: [/usr/bin/*]
```

In this case, yada will generate `${shlibs:mypackage:Depends}` variable. All `${*:mypackage:Depends}` variables will be joined to one `${mypackage:Depends}` variable.

If the "Depends" field is ommited, and the package have ELF binaries, this field will be generated automatically.

The substvars variables will be generated automatically for some of the special fields, i.e. `${doc-base:$PACKAGE:Suggests}` for "Doc-Base" field or `${menu:$PACKAGE:Suggests}` for "Menu" field.

The square brackets without filenames will be replaced by `${$PACKAGE:Field}`.

If package name contains dot (.) or colon (:), you have to replace it with hyphen (-) in shlibs variables.

More advanced example:

```
Package: mypackage
Depends: ${shlibs:mypackage:Depends}, perl5 | perl
Suggests: ${mypackage:Suggests}, www-browser
Recommends: mypackage-plugins, []

Package: mypackage-bin
Install: sh
 yada shlibdeps

Package: mypackage-libc++
Depends: ${shlibs:mypackage-libc--:Depends}, [/usr/lib/mypackage/*]
```

Provides, Conflicts, Replaces

These fields affect the interaction between packages installed on the same system. They are fully documented in the Debian packaging manual.

Description

This field is fully documented in the Debian packaging manual. If the source package paragraph contains a multi-line "Description" field, its value (apart from the first line) will be prepended to the Description field of each binary package, separated by a blank line. The following fields are defined by YADA. Often, only the "Install" field need be used.

Install

An executable field, used to build the filesystem image for the binary package. This field is called with "binary" target and "binary-indep" target for "all" architecture or "binary-arch" for other architecture.

Several extra commands are available:

yada `install`
`[-bin|-conf|-data|-dir|-doc|-game|-include|-lib|-libexec|-man|-sbin|-script|-src|-sscrip`
`[-x|-non-x] [-stripped|-unstripped] [-exec|-no-exec] [-into` *`dir`*`] [-as` *`name`*`]`
`[-subdir` *`subdir`*`] [-lang` *`lang`*`] [-section` *`mansect`*`] [-gzip|-bzip2] [-ucf]` *`file`*`...`

Install the *`file`*s named into the binary package filesystem image. There are many options to affect how the installation is done.

`-bin`

Install user binaries (into `/usr/bin`).

`-conf`

Install configuration files (into `/etc`).

`-data`

Install data files. (This is the default.)

`-dir`

Create directories in the filesystem image corresponding to each *`file`*, which should be specified as absolute pathnames on the installed destination system.

`-doc`

Install documentation files (into `/usr/share/doc/$PACKAGE`).

`-game`

Install game binaries (into `/usr/games`).

`-include`

Install include headers (into `/usr/include`).

`-lib`

Install shared libraries (into `/usr/lib`).

`-libexec`

Install additional executables (into `/usr/lib`).

`-man`

Install man pages (into `/usr/man/man?`).

`-sbin`

Install system binaries (into `/usr/sbin`).

`-script`

Install user scripts (into `/usr/bin`). It also means the same as unstripped, if there is no other file type specified.

`-src`

Install source files (into `/usr/src`).

`-sscript`

Install system scripts (into `/usr/sbin`).

`-cgi`

Install CGI files (into `/usr/lib/cgi-bin`).

`-x`

Install X-related files (they will be installed into the `/usr/include/X11` or `/usr/lib/X11` or `/usr/share/X11` hierarchy instead of into `/usr`).

`-non-x`

Install ordinary, non-X-related files (the default).

`-stripped`

Strip the files after installing them (the default for binaries and shared libraries if environment variable `DEB_BUILD_OPTIONS` matches "nostrip" string).

`-unstripped`

Do not strip (the default for everything else).

`-exec`

Make the installed files executable (the default for binaries).

`-no-exec`

Make the installed files non-executable (the default for everything else).

`-into` *`dir`*

Override the normal destination directory with *`dir`* (specified as an absolute pathname on the destination system).

`-as` *`name`*

Rename the *`file`* to *`name`* when installing it (only available when installing a single *`file`*).

`-subdir` *`subdir`*

Put the file into a subdirectory of the location it would normally be installed into.

`-lang` *`lang`*

Add another subdirectory to the location it would normally be installed into.

`-section` *`mansect`*

Install man pages into section *`mansect`*, overriding yada's normal smarts for working out the appropriate section.

`-gzip`

Compress file with **gzip** `-9` after install.

`-bzip2`

Compress file with **bzip2** after install.

`-ucf`

Use Update Configuration File (ucf) handling for configuration files. The original file will be installed into `/usr/share/ucf/`*`path`* and additional ucf calls with `--three-way` option will be used in postinst and postrm scripts. This offers a chance to see a merge of the changes between old maintainer version and the new maintainer version into the local copy of the configuration file.

yada `copy`
[`-bin`|`-conf`|`-data`|`-doc`|`-game`|`-include`|`-lib`|`-libexec`|`-man`|`-sbin`|`-script`|`-src`|`-sscript`|`-cg`
[`-x`|`-non-x`] [`-into` *`dir`*] [`-as` *`name`*] [`-subdir` *`subdir`*] [`-lang` *`lang`*] [`-section` *`mansect`*]
`file`|*`dir`*...

Copy a *`file`* or *`dir`* with preserving file attributes into the binary package filesystem image. See **yada** `install` for additional arguments.

yada `move`
[`-bin`|`-conf`|`-data`|`-doc`|`-game`|`-include`|`-lib`|`-libexec`|`-man`|`-sbin`|`-script`|`-src`|`-sscript`|`-cg`
[`-x`|`-non-x`] [`-into` *`dir`*] [`-as` *`name`*] [`-subdir` *`subdir`*] [`-lang` *`lang`*] [`-section` *`mansect`*]
`file`|*`dir`*...

Move a *`file`* or *`dir`* into the binary package filesystem image. See **yada** `install` for additional arguments.

yada `rename`
[`-bin`|`-conf`|`-data`|`-doc`|`-game`|`-include`|`-lib`|`-libexec`|`-man`|`-sbin`|`-script`|`-src`|`-sscript`|`-cg`

[-x|-non-x] [-into *dir*] [-as *name*] [-subdir *subdir*] [-lang *lang*] [-section *mansect*] *file|dir*...

Rename a *file* or *dir* in the binary package filesystem image. See **yada** install for additional arguments.

yada symlink
[-bin|-conf|-data|-doc|-game|-include|-lib|-libexec|-man|-sbin|-script|-src|-sscript|-cg
[-x|-non-x] [-into *dir*] [-as *name*] [-subdir *subdir*] [-lang *lang*] [-section *mansect*]
file|dir...

Make a symlink of *file* or *dir* as *name* in the binary package filesystem image. See **yada** install for additional arguments.

yada undocumented [-x|-non-x] [-section *mansect*] *name*...

Mark the *name*s as undocumented, by creating manpage symlinks to undocumented.7. You can either give names with the man page section appended (e.g. foo.1 or blurzle.3x) or give the section explicitly, in which case the names will not have suffixes which look like sections stripped. -x and -non-x work as for **yada** install.

yada shlibdeps [args]

This command finds the files execlutables and libraries which are dynamicaly linked. The command is called automatically, but you call use it explicity if you need to set LD_LIBRARY_PATH environment variable. All arguments will be passed to dpkg-shlibdeps(1).

yada makeshlibs [-V[*deps*]|--version[=*deps*]] [-X*item*|--exclude=*item*]

This command automatically scans for shared libraries, and generates a shlibs file for the libraries it finds.

-V[*deps*]

By default, the shlibs file generated by this command program does not make packages depend on any particular version of the package containing the shared library. It may be necessary for you to add some version dependancy information to the shlibs file. If -V is specified with no dependancy information, the current version of the package is plugged into a dependancy that looks like "package-name (>= packageversion)". If -V is specified with parameters, the parameters can be used to specify the exact dependancy information needed (be sure to include the package name).

-X*item*

Exclude files that contain *item* anywhere in their filename from being treated as shared libraries.

yada strip [-X*item*|--exclude=*item*]

This command strips executables, shared libraries, and some static libraries. It assumes that files that have names like lib*_g.a are static libraries used in debugging, and will not strip them. The command is called automatically if environment variable DEB_BUILD_OPTIONS does not match "nostrip" string and the package doesn't have "Contains: unstripped" field.

`-X`*`item`*

Exclude files that contain "item" anywhere in their filename from being stripped. You may use this option multiple times to build up a list of things to exclude.

yada `patch` *`patchfiles`*

This command applies patches that match command argument. See description of Patch field for more informations. The standalone using of yada patch command could be useful for conditional applying the patches. I.e.:

```
Build: sh
 yada patch "debian/patches/any/*"
 if [ "$DEB_BUILD_ARCH" = "i386" ]; then
     yada patch "debian/patches/i386/*"
 fi
```

The command is automatically called at the start of Build script if the Patch field is used.

yada `unpatch`

This command removes all patches previously applied by yada patch command. The command is automatically called at the end of Clean script if the Patch field is used. If not, the command have to be called explicity:

```
Clean: sh
 [...]
 yada unpatch
```

Finalise (or Finalize)

After the "Install" field is executed, all user and group ownerships in the filesystem image are set to "root", and all permissions are set to "rwxr-xr-x" for directories and for plain files which have an execute bit already set, and "rw-r--r--" for all other plain files. The Finalise executable field is used to set up any permissions or ownerships needed in the filesystem image which differ from the defaults.

The `ROOT`, `CONTROL`, `PACKAGE`, `VERSION` and `DEB_*` variables are available as in the "Install" field.

Preinst, Postinst, Prerm, Postrm, Config

These executable fields are transformed into the maintainer scripts for the binary package. Several common tasks done by maintainer scripts are prepended automatically if certain other fields are specified. The `PACKAGE` and `VERSION` variables are set for these fields.

Templates

If this field is specified, its value is placed into a control file "templates" used by debconf system. The YADA supports po-debconf(7) system, so translatable fields can be prepended with an underscore. If the file `debian/po/templates.pot` exists, the po2debconf(1) command for merging translations are called at build time.

You can update `debian/po/` directory with debconf-updatepo(1) command.

Doc-Depends

Normally, YADA creates a directory named `/usr/share/doc/`*`package`*`/` automatically and places the copyright file and changelogs in it. If the package depends on another binary package, created by the same source package, whose `/usr/share/doc/`*`package`*`/` directory is appropriate, give that package's name as the value of this field, and an appropriate symlink will be created.

Alternatives

If your package includes files to be registered using update-alternatives(8), specify them using this field. Please read the man page for update-alternatives(8) to understand the terminology in the following. Each alternative to be installed is specified by a single line.

Master links are specified by a line containing the full generic pathname, followed by the name of the symlink in the alternatives directory, followed by the full pathname of the alternative, followed by the priority of the alternative. The three names are separated by right-arrows (each made of a hyphen followed by a greater-than symbol: "->"), and the priority is surrounded by round brackets (parentheses).

Slave links are specified by a line starting with two greater-than symbols (">>"), followed by the full generic pathname, followed by the name of the slave symlink in the alternatives directory, followed by the full pathname of the alternative. The three names are separated by right-arrows (each made of a hyphen followed by a greater-than symbol: "->"). Each line describing a slave link is grouped together with the master link most recently described.

An example:

```
Alternatives:
 /usr/bin/editor -> editor -> /usr/bin/nvi (30)
 >> /usr/man/man1/editor.1.gz -> editor.1.gz -> /usr/man/man1/nvi.1.gz
```

Diversions

If your package includes files to be registered using dpkg-divert(8), specify them using this field. Please read the man page for dpkg-divert(8) to understand the terminology in the following. Each diversion to be installed is specified by a single line. The diversion is specified by a line containing the path of overriding file, followed by the path of overriden file. The two names are separated by right-arrows (each made of a hyphen followed by a greater-than symbol: "->").

An example:

```
Diversions:
 /usr/sbin/smail -> /usr/sbin/smail.real
```

Menu

If this field is specified, its value is placed into a file in the `/usr/lib/menu/` directory, and update-menus(8) is called at the appropriate moments during package installation and removal. See menufile(5) for documentation on how to write this field.

Init

This executable field is transformed into the init script placed in `/etc/init.d/` directory, and update-rc.d(8) is called at the appropriate moments during package installation and removal. The first line of this script have to contain the arguments for update-rc.d(8) command. If the init script name is ommited (the first argument), then the package name is used.

An example:

```
Init: sh
 defaults 20
 # The example init script
 #
 # The first line contains arguments for update-rc.d
 # The first argument is ommited, so the full string might be:
 # package-name defaults 20
 #
 NAME=daemon
 DAEMON=/usr/bin/daemon
 case "$1" in
   start)
     start-stop-daemon --start --quiet --pidfile /var/run/$NAME.pid \
       --exec $DAEMON;;
   stop)
     start-stop-daemon --stop --quiet --pidfile /var/run/$NAME.pid \
       --exec $DAEMON;;
 esac
```

Logrotate

If this field is specified, its value is placed into a file in the `/etc/logrotate/` directory.

Cron

If this field is specified, its value is used as system-wide crontab file in the `/etc/cron.d/` directory.

Cron-Hourly, Cron-Daily, Cron-Weekly, Cron-Monthly

These executable fields are transformed into the system-wide crontab scripts for the binary package. These scripts will be installed into `/etc/cron.{hourly,daily,weekly,monthly}` directories.

Modutils

If this field is specified, its value is placed into a file in the `/etc/modutils/` directory.

Pam

If this field is specified, its value is placed into a file in the `/etc/pam.d/` directory.

Shlibs

If this field is specified, its value is used as the contents of the package's "shlibs" control area file.

Contains

This field controls additional calls from maintainer's scripts. The value is a list of tags:

libs

Assumes the package contains shared libraries, and calls ldconfig(8) at the appropriate point during package installation.

unstripped

Assumes the package contains unstripped binaries, so it is not stripped automatically by **yada** `strip`.

xfonts

Assumes the package contains X Window System's fonts, and calls update-fonts-alias(8) and update-fonts-scale(8).

kernel-modules

Assumes the package contains kernel modules, and this package can be used with make-kpkg(1) utility. It also doesn't strip binaries.

An example:

```
Contains: libs
```

Overrides

Lintian is a Debian package checker which generates information about policy violations of package. The format of Lintian's output is:

```
X: package: full information about violation
```

i.e.:

```
W: securecgi: setuid-binary usr/lib/cgi-bin/securecgi 4755 root/root
```

The Overrides field allows to ignore some Lintian's messages. In this example, to ignore above message it is required to put the Overrides field at binary section:

```
Package: securecgi
Overrides: setuid-binary usr/lib/cgi-bin/securecgi 4755 root/root
```

5. Macro preprocessor

Macro preprocessor resolves all macro commands and macro variables used in `debian/packages`file and produces a temporary file `debian/packages-tmp`.

Macro commands begin with percent mark ("%"):

%define

Definition of macro variable. This variable can be used further in common section or another macro command.

%include

The preprocessor will include another file specified as parameter for this macro command.

%if, %else, %endif

These macro commands are used to make conditional skipping. The first should be followed by text. If the condition text is not equal 0 or is not an empty string then the condition is true. The conditional macro commands can be nested.

Macro variables:

%{MACRO_VAR}

Expands to variable predefined with "%define" command or generates warning if the variable is not defined already.

%{$ENV_VAR}

Expands to environment variable. If variable is not defined, expands to empty string.

%{?MACRO_VAR:string}

Expands to given string if macro variable is set and its value is true.

%{!?MACRO_VAR:string}

Expands to given string if macro variable is not defined or its value is not true.

%{?$ENV_VAR:string}

Expands to given string if environment variable is set and its value is true.

%{!?$ENV_VAR:string}

Expands to given string if environment variable is not defined or its value is not true.

%‘command‘

Executes given command and expands to its output.

There are following predefined macro variables:

`VERSION`

Defines the version of the source package.

`SOURCE`

Defines the name of the source package.

`YADA_COMMAND`

Defines how YADA program is called.

`YADA_VERSION`

Defines YADA version. Internal version name is overriden by `$YADA_VERSION` environment variable. This setting it is used in generated Build-Depends field from `debian/control`file.

There are following special macro variables:

`with_*`, `without_*`

If %{with_*} macro is set, then %{without_*} macro is unset. If %{without_*} macro is set, then %{with_*} macro is unset. If `$YADA_WITH_*` environment variable is set, it is used as %{with_*} macro variable, and its name is converted from uppercase to lowercase. If `$YADA_WITHOUT_*` environment variable is set, it is used as %{without_*} macro variable, and its name is converted from uppercase to lowercase.

`DEB_BUILD_*`, `DEB_HOST_*`

These macro variables are set based on dpkg-architecture(1) command output.

`with_DEB_BUILD_*`, `without_DEB_BUILD_*`, `with_DEB_HOST_*`, `without_DEB_HOST_*`

These macro variables are set based on dpkg-architecture(1) command output. If, i.e. %{DEB_HOST_ARCH} is set to "i386", then %{with_DEB_HOST_ARCH_i386} macro is set.

Macro variables can be nested.

The example usage of macro preprocessor:

```
%define KSRC %{?$KSRC:%{$KSRC}}%{!?$KSRC:/usr/src/linux}
%define KVERS %{?$KVERS:%{$KVERS}}%{!?$KVERS:%`sed -n -e '/UTS_RELEASE/s/^[^"]*"\([^"]*\
%define KDREV %{?$KDREV:%{$KDREV}}%{!?$KDREV:UNKNOWN}
%define APPEND_TO_VERSION %{$APPEND_TO_VERSION}
%define FLAVOUR %{$FLAVOUR}
%define KMAINT %{?$KMAINT:%{$KMAINT}}%{!?$KMAINT:%{$DEBFULLMAIL}}
%define KEMAIL %{?$KEMAIL:%{$KEMAIL}}%{!?$KEMAIL:%{$DEBEMAIL}}

Source: foo-modules-source
Build-Depends: yada (>= %{YADA_VERSION})
[...]

Package: foo-modules-%{KVERS}
Architecture: any
%if %{KDREV}
Recommends: kernel-image-%{KVERS} (= %{KDREV})
%else
Recommends: kernel-image-%{KVERS}
%endif
Description: Some foo kernel modules
 The example usage of YADA macro preprocessor.
Contains: unstripped
Install: sh
%if %{with_foo}
 yada install -lib -unstripped -into /lib/modules/%{KVERS}/kernel/foo src/foo.o
%endif
[...]
```

6. FAQ

Q:

How to make a backport for a package which uses YADA?

A:

If YADA is not available for older distribution release, copy the `/usr/bin/yada` into `debian/` subdirectory and call **debian/yada** `rebuild`.

Q:

Can I export DebConf templates or other script to external file?

A:

You can use macro commands. Example `debian/package` file:

```
Package: mypackage
Depends: otherpackage, []
Install:
 # some installing commands
Templates:
%`sed -e 's/^$/./' -e 's/^/ /' debian/packages.templates`
# no space in first row!
Config: sh
[...]
```

Example debian/package.templates file:

```
Template: %{PACKAGE}/debconf-template
Type: note
Description: DebConf template in external file
 As you might see, you can use macros in the template file.
```

Don't use `debian/templates` file. This file is automatically created based on `debian/packages`file.

Q:

How YADA supports po-debconf?

A:

Run **yada** `rebuild templates` and then **debconf-gettextize** or **debconf-updatepo**. Delete `debian/templates` file after all.

Q:

Is there any syntax highlighting for `debian/packages`?

A:

There is a file prepared for Midnight Commander. You can find `debian-control.syntax` file in documentation directory, then put it into `~/.mc/cedit` directory. Make sure there is a line in `~/.mc/cedit/Syntax` file:

```
file (control|packages)$ Debian\scontrol\sfile
include debian-control.syntax
```